CAD via Projection and Lifting (V3.17)
Matthew England (University of Bath)
11th February 2015

```
> restart: kernelopts(opaquemodules=false):
> Digits:=20:
```

This file is an introduction to the Maple package "ProjectionCAD". (This iteration of the code is substantially different to the last, hence labeled V3).
The code uses and mimics work in the RegularChains library. Hence it is useful (but not necessary) to load this.

```
> with(RegularChains): with(SemiAlgebraicSetTools):
```

The code is designed for and has been tested in Maple 18 (and the RegularChains Library this ships with).
It is very likely to work with later versions of Maple, as well as Maple 15-17, but has not been tested there.
It will not work with Maple 14 or earlier as the RegularChains commands used were not present. However, if you have such an earlier Maple you could try installing the RegularChains Library from www.regularchains.org.

```
> read("ProjectionCAD.mpl"):
  with(ProjectionCAD);
```

"This is V3.18 of the ProjectionCAD module from 11th February 2015, designed and tested for use in Maple 18."

[*CADDist*, *CADFull*, *CADGenerateStack*, *CADLifting*, *CADNormDist*, *CADProjection*, *ECCAD*, *ECCADFormulations*, *ECCADHeuristic*, *ECCADProjFactors*, *ECCADProjOp*, *LCAD*, *LCADDisplay*, *LCADRecursive*, *LTTICAD*, *LVCAD*, *LVTTICAD*, *NumCellsInCAD*, *NumCellsInPiecewiseCAD*, *TTICAD*, *TTICADDist*, *TTICADFormulations*, *TTICADHeuristic*, *TTICADNormDist*, *TTICADProjFactors*, *TTICADProjOp*, *TTICADQFFFormulations*, *TTICADQFFHeuristic*, *TTICADResCAD*, *TTICADResCADSet*, *VCAD*, *VCADLiftOverLowCAD*, *VTTICAD*, *VariableOrderingHeuristic*, *VariableOrderings*, *ndrr*, *sotd*] **(1)**

The code in this module is an implementation of algorithms for constructing various CADs via projection and lifting. This includes CADs that are: sign-invariant, order-invariant, sign-invariant over an equational constraint and truth-table invariant. There are also tools for considering the different formulations for these algorithms and heuristics for making these choices. The latest addition is code to produce sub-CADs containing cells of a given dimension or lying on a given variety.

We note that there is already a command in Maple to produce CADs:
RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose
However, this uses a different approach (triangular decomposition). Indeed, we named our package to contrast with it.

We introduce the main commands of our module in the sections below. Note that details for an individual command can be obtained using **Describe**. For example:

```
> Describe(NumCellsInPiecewiseCAD);

# NumCellsInPiecewiseCAD: Calculate the number of cells in a CAD
given in the
# piecewise output format. Note that this command will not work
```

```
for sub-CADs.
# Input: A CAD in piecewise format. This could be output from
either this
# module or the Regular Chains implementation.
# Output: The number of cells in the CAD.
NumCellsInPiecewiseCAD( pwcad::piecewise, $ ) :: posint
```

CONTENTS:

0. Note on known bug in Maple 18.

There is a known bug in Maple 18 which affects ProjectionCAD.
The bug concerns the **RootOf(p,a,b)** command. This represents algebraic numbers as the root of a polynomial p between rational numbers a<b.
In Maple 18 if we have a<b but evalf(a)=evalf(b) under the current precision we obtain an error message. The error can be avoided by increasing the precision.
Example:

```
> restart: with(RegularChains): with(SemiAlgebraicSetTools): read
  ("ProjectionCAD.mpl"): with(ProjectionCAD):
```

"This is V3.17 of the ProjectionCAD module from 10th February 2015, designed and tested for use in Maple 18." **(1.1)**

```
> f1:=x^2+y^2+z^2-1: g1:=x*y*z-1/4:
  f2:=x^2-y^2-z^2-1: g2:=(x-4)*(y-1)-1/4+z:
  PHI:=[ [f1,[g1]], [f2,[g2]] ]:
> Digits;
  ord:=[z,y,x]: TTICAD( PHI, ord): ord,nops(%);
```

10

Error, (in RootOf) the input range -189907187703/137438953472 .. -379814375405/274877906944 is invalid

$[z, y, x], 3$ **(1.2)**

```
> Digits:=20;
  ord:=[z,y,x]: TTICAD( PHI, ord): ord,nops(%);
```

$Digits := 20$

$[z, y, x], 463$ **(1.3)**

```
> Digits:=10:
```

The bug is not present in prior versions of Maple.
The bug has been reported to Maplesoft, should not be present in future versions of Maple (and is not present in the Maple 2015 Beta).
In the remainder of this worksheet the Digits are increased when required to avoid the bug.

1. Sign-invariant CADs

Given polynomials and a variable ordering **CADFull** will construct a sign-invariant CAD using either Collins or McCallum projection (default is McCallum) and the corresponding lifting algorithm.

For example:

```
> f:=y^3+y^2+y*x^2-1;
```

$$f := y\,x^2 + y^3 + y^2 - 1 \tag{2.1}$$

```
> CADFull( {f}, [y,x], method=Collins): nops(%);
  CADFull( {f}, [y,x], method=McCallum): nops(%);
```

$$15$$
$$3 \tag{2.2}$$

The projection and lifting steps may be performed separately if desired. The projection factors may be calculated using:

```
> psetC:=CADProjection( {f}, [y,x], method=Collins );
  psetM:=CADProjection( {f}, [y,x], method=McCallum );
```

$$psetC := \{x^2 + 1,\ 3\,x^2 - 1,\ 4\,x^4 - 5\,x^2 + 23,\ y^3 + y\,x^2 + y^2 - 1\}$$
$$psetM := \{x^2 + 1,\ 4\,x^4 - 5\,x^2 + 23,\ y^3 + y\,x^2 + y^2 - 1\} \tag{2.3}$$

Then we can lift with respect to these using:

```
> CADLifting( psetC, [y,x], method=Collins): nops(%);
```

$$15 \tag{2.4}$$

Note that if you were to lift a set of projections polynomials from one algorithm with the other then the output is no longer guaranteed.

2. CAD output formats

There are currently four style of output format for CADs: ***list***, ***listwithrep***, ***rootof*** and ***piecewise***.
The default is list, which gives a list of cells, each containing an index and a samplepoint.
Note that this is the same format as RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose with output=list.

```
> f:=y^3+y^2+y*x^2-1: cadl:=CADFull( {f}, [y,x], method=McCallum,
  output=list ); nops(%);
```

$$cadl := \left[[[1,1],[regular\_chain,[[0,0],[-1,-1]]]],\left[[1,2],\left[regular\_chain,\left[[0,0],\left[\frac{3}{4},1\right]\right]\right]\right],[[1,3],[regular\_chain,[[0,0],[2,2]]]]\right]$$

3 **(3.1)**

The choice output=rootof will give a list of cells, each represented by conditions on the variables.
Note that this is the same format as RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose with output=rootof.

```
> cadr:=CADFull( {f}, [y,x], method=McCallum, output=rootof );
  nops(%);
```

$$cadr := \left[\left[x=x, y<RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right],\left[x=x, y=RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right],\left[x=x, RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)<y\right]\right]$$

3 **(3.2)**

The choice output=listwithrep will give a list of cells which contain all the information from the previous two formats.

```
> cadlwr:=CADFull( {f}, [y,x], method=McCallum, output=
  listwithrep ); nops(%);
```

$$cadlwr := \left[\left[[1,1],\left[x=x, y<RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right]\right],[regular\_chain,[[0,0],[-1,-1]]]\right],\left[[1,2],\left[x=x, y=RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)\right],\left[regular\_chain,\left[[0,0],\left[\frac{3}{4},1\right]\right]\right]\right],\left[[1,3],\left[x=x, RootOf\left(\_Z^3+\_Z x^2+\_Z^2-1, index=real_1\right)<y\right],[regular\_chain,[[0,0],[2,2]]]\right]\right]$$

3 **(3.3)**

The choice output=piecewise will show the CAD arranged cylindrically using Maple's piecewise construction.

```
> cadp:=CADFull( {f}, [y,x], method=McCallum, output=piecewise );
```

(3.4

$$cadp := \begin{cases} [regular\_chain, [[0,0],[-1,-1]]] & y < RootOf\left(\_Z^3 + \_Z x^2 + \_Z^2 - 1, index = real_1\right) \\ \left[regular\_chain, \left[[0,0], \left[\frac{3}{4}, 1\right]\right]\right] & y = RootOf\left(\_Z^3 + \_Z x^2 + \_Z^2 - 1, index = real_1\right) \\ [regular\_chain, [[0,0],[2,2]]] & RootOf\left(\_Z^3 + \_Z x^2 + \_Z^2 - 1, index = real_1\right) < y \end{cases} \quad \textbf{(3.4}$$

This format is comparable with RegularChains:-SemiAlgebraicSetTools:-CylindricalAlgebraicDecompose with output=piecewise, however, here radicals are used instead of RootOf constructions for easier examples.

```
> CADFull( {x^2+y^2-1}, [y,x], method=McCallum, output=piecewise
  );
```

$$\begin{cases} [regular\_chain, [[-2,-2],[0,0]]] & x < -1 \\ \begin{cases} [regular\_chain, [[-1,-1],[-1,-1]]] & y < 0 \\ [regular\_chain, [[-1,-1],[0,0]]] & y = 0 \\ [regular\_chain, [[-1,-1],[1,1]]] & 0 < y \end{cases} & x = -1 \\ \begin{cases} [regular\_chain, [[0,0],[-2,-2]]] & y < -\sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[-1,-1]]] & y = -\sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[0,0]]] & -\sqrt{-x^2+1} < y < \sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[1,1]]] & y = \sqrt{-x^2+1} \\ [regular\_chain, [[0,0],[2,2]]] & \sqrt{-x^2+1} < y \end{cases} & -1 < x < 1 \\ \begin{cases} [regular\_chain, [[1,1],[-1,-1]]] & y < 0 \\ [regular\_chain, [[1,1],[0,0]]] & y = 0 \\ [regular\_chain, [[1,1],[1,1]]] & 0 < y \end{cases} & x = 1 \\ [regular\_chain, [[2,2],[0,0]]] & 1 < x \end{cases} \quad \textbf{(3.5)}$$

Note that we cannot measure the number of cells in a piecewise cad using the nops functions. Use the function ***CADNumCellsInPiecewise*** instead

```
> nops(cadp); NumCellsInPiecewiseCAD(cadp);
```

$$6$$
$$3 \quad \textbf{(3.6)}$$

3. Examing failure (userinfo levels)

Collins' algorithm will always succeed (given sufficient time and memory).
McCallum's algorithm can fail when the input polynomials are not well oriented.

McCallum's algorithm has been adapted here to maximise success by guaranteeing only a sign-invariant CAD (rather than order-invariant) as a final output.
Hence to give an example of failure we need at least 5-dimensions .
(This is because 2d is always OI, 3d can only fail on dim zero cell for which we have McCallum's delineating polynomial method and 4d can still be guaranteed sign-invariant.)
A trivial example of failure:

```
> f:=a*e+b*d+c*e+d+e;
```

$$f := a\,e + b\,d + c\,e + d + e \tag{4.1}$$

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ): nops(%);
Warning, The input is not well-oriented (there is
nullification on cell [1, 2, 2]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [2, 2, 1]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [3, 4, 2]).  The output cannot be
guaranteed correct.
```

241 **(4.2)**

In this case the algorithm continued, but the warning implies that the output may be meaningless.
We can change what happens in this situation with an optional argument.
Using failure=warn gives the output above. Using failure=err will generate an error message.
Using failure=giveFAIL will return the boolean value FAIL.

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum, failure=err );
Error, (in PCAD ProjCADLift) The input is not well-oriented
(there is nullification on cell [1, 2, 2]).  The output cannot
be guaranteed correct.
```

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum, failure=giveFAIL );
```

FAIL **(4.3)**

```
> CADFull( {f}, [a,b,c,d,e], method=McCallum, failure=warn ):
Warning, The input is not well-oriented (there is
nullification on cell [1, 2, 2]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [2, 2, 1]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [3, 4, 2]).  The output cannot be
guaranteed correct.
```

The default is to warn if the output is guarenteed sign-invariant but not order invariant and to give FAIL otherwise. Note that if you allow the algorithm to keep running in such cases it may get in an infinite loop.

Note that we can of course get a guaranteed correct output in all cases by using Collins:

```
> CADFull( {f}, [a,b,c,d,e], method=Collins  ): nops(%);
```

241 **(4.4)**

In this case the number of cells in Collins guaranteed SI CAD was the same as in the "failed" McCallum cad.

UserInfo
We can get more information of the algorithm CADFull as it runs by changing the userinfo level of ProjectionCAD.
Level 1 is the default and gives no information other than errors and warnings.
Level 2 will also report on the progress of the algorithm as it runs and give details on reasons for failure.
Level 3 will also list the all the cells where nullification occurs and what happened because of it.
There is also Level 4 which is rarely used. It sometimes prints the polynomials in question.

Comparing user info levels

```
> infolevel[ProjectionCAD]:=1:
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ):
Warning, The input is not well-oriented (there is
nullification on cell [1, 2, 2]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [2, 2, 1]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [3, 4, 2]).  The output cannot be
guaranteed correct.
> infolevel[ProjectionCAD]:=2:
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ):
CADFull: produced set of 6 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [e] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [d e] -space with 13 cells
PCAD_ProjCADLift: produced CAD of [c d e] -space with 33
cells
Warning, The input is not well-oriented (there is
nullification on cell [1, 2, 2]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [2, 2, 1]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [3, 4, 2]).  The output cannot be
guaranteed correct.
PCAD_ProjCADLift: produced CAD of [b c d e] -space with 85
cells
PCAD_ProjCADLift: produced CAD of [a b c d e] -space with
241 cells
> infolevel[ProjectionCAD]:=3:
> CADFull( {f}, [a,b,c,d,e], method=McCallum  ):
CADFull: produced set of 6 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [e] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [d e] -space with 13 cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2]
PCAD_ProjCADLift: the nullified polynomial was discarded
```

```
since cell [2 2] is zero-dim and the minimal delineating
polynomial is a constant.
PCAD_ProjCADLift: produced CAD of [c d e] -space with 33
cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [1 2 2]
PCAD_ProjCADLift: there is nullification on cell [1 2 2]
which has dim>0. Hence the CAD cannot be guaranteed sign-
invariant.
Warning, The input is not well-oriented (there is
nullification on cell [1, 2, 2]).  The output cannot be
guaranteed correct.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2 1]
PCAD_ProjCADLift: there is nullification on cell [2 2 1]
which has dim>0. Hence the CAD cannot be guaranteed sign-
invariant.
Warning, The input is not well-oriented (there is
nullification on cell [2, 2, 1]).  The output cannot be
guaranteed correct.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [3 4 2]
PCAD_ProjCADLift: there is nullification on cell [3 4 2]
which has dim>0. Hence the CAD cannot be guaranteed sign-
invariant.
Warning, The input is not well-oriented (there is
nullification on cell [3, 4, 2]).  The output cannot be
guaranteed correct.
PCAD_ProjCADLift: produced CAD of [b c d e] -space with 85
cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 1 1 2]
PCAD_ProjCADLift: the nullified polynomial was discarded
since this is the final lift and only sign invariance is
required.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2 1 1]
PCAD_ProjCADLift: the nullified polynomial was discarded
since this is the final lift and only sign invariance is
required.
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 3 1 2]
PCAD_ProjCADLift: the nullified polynomial was discarded
since this is the final lift and only sign invariance is
required.
PCAD_ProjCADLift: produced CAD of [a b c d e] -space with
241 cells
```

Note that userinfo statements can significantly increase the time taken by algorithms and so shouldn't be used as default.

```
> infolevel[ProjectionCAD]:=1:
```

Note that similar userinfo statements are available for the algorithms ECCAD and TTICAD discussed later.

4. Order invariant CADs

As discussed above, McCallum's CADW algorithm has been modified to maximise success - this is the default.
To run McCallum's original algorithm, guaranteeing order-invariance at the final step, (or failure), then the optional argument finalCAD=OI must be given.
Consider for example:

```
> f:=x^2-z*y;
```

$$f := x^2 - z\,y \tag{5.1}$$

```
> infolevel[CADFull]:=3:
> cad1:=CADFull( {f}, [z,y,x], method=McCallum, output=
  listwithrep  ): nops(%);
CADFull: produced set of 3 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [x] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [y x] -space with 9 cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2]
PCAD_ProjCADLift: the nullified polynomial was discarded since
this is the final lift and only sign invariance is required.
PCAD_ProjCADLift: produced CAD of [z y x] -space with 21 cells
```

$$21 \tag{5.2}$$

```
> cad2:=CADFull( {f}, [z,y,x], method=McCallum, finalCAD=OI,
  output=listwithrep  ): nops(%);
CADFull: produced set of 3 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [x] -space with 3 cells
PCAD_ProjCADLift: produced CAD of [y x] -space with 9 cells
PCAD_ProjCADLift: a projection polynomial was nullified on
cell [2 2]
PCAD_ProjCADLift: cell [2 2] is zero-dim so a delineating
polynomial was used.
PCAD_ProjCADLift: produced CAD of [z y x] -space with 23 cells
```

$$23 \tag{5.3}$$

The first cad only has 21 cells, the second 23. Look at the difference by considering the indices

```
> convert( map(X->op(1,X), cad2),set) minus convert(map(X->op(1,
  X),cad1),set);
```

$$\{[2, 2, 2], [2, 2, 3]\} \tag{5.4}$$

```
> select(X->op(1,X)=[2,2,1], cad2);
  select(X->op(1,X)=[2,2,2], cad2);
  select(X->op(1,X)=[2,2,3], cad2);
```

$$[[[2, 2, 1], [x = 0, y = 0, z < 0], [regular\_chain, [[0, 0], [0, 0], [-1, -1]]]]]$$
$$[[[2, 2, 2], [x = 0, y = 0, z = 0], [regular\_chain, [[0, 0], [0, 0], [0, 0]]]]]$$
$$[[[2, 2, 3], [x = 0, y = 0, 0 < z], [regular\_chain, [[0, 0], [0, 0], [1, 1]]]]] \tag{5.5}$$

```
> select(X->op(1,X)=[2,2,1], cad1);
```

$$[[[2, 2, 1], [x = 0, y = 0, z = z], [regular\_chain, [[0, 0], [0, 0], [0, 0]]]]] \tag{5.6}$$

The difference occurs when x=0 and y=0. The first cad has just one cell here, with z free. The second splits into three cells, with z<0,z=0 and z>0.
This is clearly needed for OI, since the order of f at zero is 2 while the order elsewhere is 1 or 0.

By considering the user info statements we see why the outputs differ. In both cases there is a nullification at the final lift.
The first time this was ignored since it was the final lift and it was assumed only a sign-invariant cad is required.
The second time this nullification was examined. Since the cell in question was zero-dimensional we could use McCallum's delineating polynomial method to guarantee order invariance.

```
> infolevel[CADFull]:=1:
```

5. Generate Stack and returning induced CADs

Part of the lifting algorithm involves building a stack over a cell such that a set of polynomials are delineable.
To do this we use the internal RegularChains algorithms, ***RegularChains:-TRD_generate_stack***.
The polynomials passed to this algorithm should separate above the cell, hence to ensure correctness, we must first process our polynomials so that this is true.
The user algorithm, ***CADGenerateStack*** first error checks, then does this pre-processing before finally passing to the TRD algorithm.
Consider for example:

```
> f:=z^2+y^2+x^2-1;
  fullcad:=CADFull([f], [z,y,x], method=McCallum): nops(%);
```

$$f := x^2 + y^2 + z^2 - 1$$

$$25 \quad \textbf{(6.1)}$$

We can force CADFull to break halfway and return the cad of over (x,y)-space using the optional argument retcad=i

```
> xycad:=CADFull([f], [z,y,x], method=McCallum, retcad=2): nops
  (%);
```

$$13 \quad \textbf{(6.2)}$$

Consider a cell of this cad.

```
> cell:=xycad[3];
```

$$cell := [[2, 2], [regular\_chain, [[-1, -1], [0, 0]]]] \quad \textbf{(6.3)}$$

If we generate the stack over this cell wrt to f, we get a list of new cells.

```
> CADGenerateStack( cell, [f], [z,y,x] );
```

$$[[[2, 2, 1], [regular\_chain, [[-1, -1], [0, 0], [-1, -1]]]], [[2, 2, 2], [regular\_chain, [[-1, -1], [0, 0], [0, 0]]]], [[2, 2, 3], [regular\_chain, [[-1, -1], [0, 0], [1, 1]]]]] \quad \textbf{(6.4)}$$

Note that each cell has an index which starts with the index of the previous cell.
If we were to do this to all cells and put them together we get the full cad:

```
> cad:=[]:
  for i from 1 to 13 do
    cad:=[op(cad), op(CADGenerateStack(xycad[i],[f],[z,y,x]))]:
  od: nops(cad);
```

$$25 \quad \textbf{(6.5)}$$

The two cads can be shown to be identical under inspection

6. CADs with equational constraint

We can build CADs which are sign-invariant with respect to an equational constraint. For example, consider

```
> f:=x^2+y^2-1:
  g:=(x-1/2)*(y-1/2):
```

and the problem f=0, g>0. We can use ECCAD to build the CAD, the first argument is a designated equational constraint and the second a list of any other constraints.

```
> ECCAD( [f, [g]], [y,x]): nops(%);
```

$$43 \tag{7.1}$$

Note that this CAD has less cells than a full sign-invariant CAD.

```
> CADFull( [f,g], [y,x], method=McCallum): nops(%);
```

$$61 \tag{7.2}$$

The algorithm follows McCallum's approach, using his reduced projection operator for the first projection. Hence there is no method choice here.

The algorithm only makes use of one equational constraint. If a problem has more than one then a choice must be made for which to designate.

For example, consider the previous problem with the extra constraint h=0 where

```
> h:=x*y-1:
```

Then we can can obtain a list of the different formulations using ECCADFormulations on a list of equational constraints and a list of non-equational constraints.

```
> Forms:=ECCADFormulations( [[f,h], [g]] );
```

$$Forms := \left[\left[y^2+x^2-1, \left[x\,y-1, x\,y-\frac{1}{2}\,x-\frac{1}{2}\,y+\frac{1}{4}\right]\right], \left[x\,y-1, \left[y^2+x^2-1, x\,y-\frac{1}{2}\,x-\frac{1}{2}\,y+\frac{1}{4}\right]\right]\right] \tag{7.3}$$

In this example, a smaller CAD is obtained by designating h:

```
> ECCAD( Forms[1], [y,x]): nops(%);
  ECCAD( Forms[2], [y,x]): nops(%);
```

$$43$$
$$19 \tag{7.4}$$

We can attempt to predict this using heuristics which examine the projection factors.

The projection factors can be obtained using ECCADProjFactors, which takes the same input as ECCAD.

NOTE: The projection polynomials are all those used in Projection. However, unlike CADFull, they are not ALL used in Lifting. In particular the non-equational constraints are not usually used in the final lift.

For more information on this see the section of projection / lifting polynomials below.

```
> PF1:=ECCADProjFactors( Forms[1], [y,x]);
  PF2:=ECCADProjFactors( Forms[2], [y,x]);
```

$$PF1 := \left\{x-1, x+1, x-\frac{1}{2}, y-\frac{1}{2}, x^2-\frac{3}{4}, y\,x-1, x^4-x^2+1, y^2+x^2-1\right\}$$
$$PF2 := \left\{x, x-2, x-\frac{1}{2}, y-\frac{1}{2}, y\,x-1, x^4-x^2+1, y^2+x^2-1\right\} \tag{7.5}$$

We use the measures sotd (sum of total degree of all factors) and ndrr (number of distinct real roots of univariate factors) to estimate CAD complexity.

```
> [sotd(PF1), sotd(PF2)];
```

$$[18, 16] \tag{7.6}$$

```
> [ndrr(PF1,x), ndrr(PF2,x)];
```

$$[5, 3] \tag{7.7}$$

We see that both predict designating h gives a less complicated CAD.

These steps can be performed by the command ECCADHeuristic (taking the input of ECCADFormulations and the variable ordering)

```
> ECCADHeuristic( [[f,h], [g]], [y,x] );
```

$$\left[y\,x-1, \left[y^2+x^2-1, \left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)\right]\right] \tag{7.8}$$

By default this picks the formulation with lowest ndrr, breaking ties with sotd (NS). However, the reverse (SN), using one only (S, N), or a weighted average of the relative heuristics (W) are also available and specified by heuristic.

```
> ECCADHeuristic( [[f,h], [g]], [y,x], heuristic=S );
```

$$\left[y\,x-1, \left[y^2+x^2-1, \left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)\right]\right] \tag{7.9}$$

```
> ECCADHeuristic( [[f,h], [g]], [y,x], heuristic=W );
```

$$\left[y\,x-1, \left[y^2+x^2-1, \left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)\right]\right] \tag{7.10}$$

See the TTICAD section for more details on heuristics commands.

7. Truth table invariant CAD

The command TTICAD produces a CAD which is truth table invariant for a list of QFFs (quantifier free formulas). Each QFF is represented by an equational constraint and a list of other constraints. For example, consider.

```
> f1:=x^2+y^2-1:
  g1:=x*y-1/4:
  f2:=(x-4)^2+(y-1)^2-1:
  g2:=(x-4)*(y-1)-1/4:
```

and the formula PHI = phi[1] ∨ phi[2]
where
phi[1] = { f1=0, g1<0 }
phi[2] = { f2=0, g2<0 }.

We treat phi[2] and phi[2] as different QFFs and so use the input

```
> PHI:=[ [f1,[g1]], [f2,[g2]] ]:
```

Then we find the TTICAD with respect to a given variable ordering

```
> TTICAD( PHI, [y,x] ): nops(%);
```

105 **(8.1)**

Compare this to a sign-invariant CAD:

```
> CADFull( [f1,f2,g1,g2], [y,x], method=McCallum): nops(%);
```

317 **(8.2)**

or the CAD that could be constructed with an implicit equational constraint

```
> ECCAD( [f1*f2, [f1,g1,f2,g2]], [y,x] ): nops(%);
```

145 **(8.3)**

We can use the retcad option to see the difference in the induced 1d cads

```
> CADFull( [f1,f2,g1,g2], [y,x], method=McCallum, retcad=1): nops
  (%);
  ECCAD( [f1*f2, [f1,g1,f2,g2]], [y,x], retcad=1 ): nops(%);
  TTICAD( PHI, [y,x], retcad=1 ): nops(%);
```

41
33
25 **(8.4)**

We see that a small difference in cells (41-25=16) is magnified to a larger difference (317-105-212) after lifting.

We can see the set of projection factors produced using the command TTICADProjFactors on TTICAD input. Note that these are the polynomials used in the projection but they may not all be used for lifting (see Section 10).

```
> P:=TTICADProjFactors( PHI, [y,x]);
```

$$P := \left\{x-5,\ x-3,\ x-1,\ x+1,\ yx-\frac{1}{4},\ x^2-4x+\frac{285}{68},\ x^4-x^2+\frac{1}{16},\ yx-x-4y\right. \quad \textbf{(8.5)}$$
$$\left.+\frac{15}{4},\ x^4-16x^3+95x^2-248x+\frac{3841}{16},\ y^2+x^2-1,\ y^2+x^2-2y-8x+16\right\}$$

TTICAD offers the same output formats as a sign-invariant CAD, discussed above.

```
> TTICAD( [ [x+1,[x,x^2]], [x-1,[x+2*x]] ], [x], output=
  piecewise); NumCellsInPiecewiseCAD(%);
```

$$\begin{cases} [regular\_chain, [[-2, -2]]] & x < -1 \\ [regular\_chain, [[-1, -1]]] & x = -1 \\ [regular\_chain, [[0, 0]]] & -1 < x < 1 \\ [regular\_chain, [[1, 1]]] & x = 1 \\ [regular\_chain, [[2, 2]]] & 1 < x \end{cases}$$

5 **(8.6)**

When input is given containing only one variable then a sign-invariant CAD is created for the equational constraints only, as in the example above.

Of course, the variable ordering is important:

```
> TTICAD( PHI, [x,y] ): nops(%);
```

153 **(8.7)**

We note that the TTICAD theory can extend to a sequence of clauses in which not all have an equational constraint.

For example, consider the problem above but with f1<0 instead of f1=0.

The implicit equational constraint approach cannot be used but TTICAD can, still giving a reduction compared to a full cad:

```
> PHI2:=[ [[],[f1,g1]], [f2,[g2]] ]:
  TTICAD( PHI2, [y,x] ): nops(%);
```

183 **(8.8)**

TTICAD will offer a benefit over a full cad unless there are no equational constraints at all

```
> PHI3:=[ [[],[f1,g1]], [[],[f2,g2]] ]:
  TTICAD( PHI, [y,x] ): nops(%);
```

105 **(8.9)**

Finally, we note that if there is more than one equational constraint then the TTICAD algorithm will choose which to use based on the heuristics discussed below.

8. The ResCAD algorithm for TTICAD

We can also attempt to create a TTICAD using the ResCAD approach. This is a conceptually simpler algorithm but not as widely applicable.
There is no practical reason to use this approach here, since TTICAD is superior. It is included to compare for experimentation and to compare with other systems.

The ResCAD set can be calculated using:

```
> rcs:=TTICADResCADSet( PHI, [y,x] );
```

$$rcs := \{16\,x^4 - 16\,x^2 + 1,\ y^2 + x^2 - 1,\ 16\,x^4 - 256\,x^3 + 1520\,x^2 - 3968\,x + 3841,\ y^2 + x^2 - 2\,y - 8\,x + 16\} \quad (9.1)$$

Providing no EC is nullified, a full CAD of this set using McCallum projection will be a TTICAD

```
> CADFull( rcs, [y,x], method=McCallum, output=list): nops(%);
```

$$105 \quad (9.2)$$

This approach is combined into one command:

```
> TTICADResCAD( PHI, [y,x] ): nops(%);
```

```
Warning, The output from TTICADResCAD is only guaranteed truth table invariant if no equational constraint is nullified by a point in [x]
```

$$105 \quad (9.3)$$

We gives some examples where the ResCAD approach can fail

Example 1 - An equational constraint is reducible:

Consider

```
> f:=x*y: g:=x+y+1: PHI:=[ [f,[g]] ];
```

$$PHI := [[y\,x, [x + y + 1]]] \quad (9.4)$$

```
> rescad:=TTICADResCAD( PHI, [y,x], output=piecewise ):
  NumCellsInPiecewiseCAD(%);
```

```
Warning, The output from TTICADResCAD is only guaranteed truth table invariant if no equational constraint is nullified by a point in [x]
```

$$15 \quad (9.5)$$

Note there is a cell [x=0, y<0] over which f is zero but g changes sign. We can avoid this error by using the full TTICAD algorithm

```
> TTICAD( PHI, [y,x], output=piecewise): NumCellsInPiecewiseCAD
  (%);
```

$$17 \quad (9.6)$$

The TTICAD splits this cell, but this is still smaller than the sign-invariant CAD

```
> CADFull( [f,g], [y,x], output=piecewise, method=McCallum):
  NumCellsInPiecewiseCAD(%);
```

$$23 \quad (9.7)$$

Example 2 - An equational constraint is nullified:

Consider

```
> f:=z*y-x^2: g:=x+y+z-1: PHI:=[ [f,[g]] ];
```

$$PHI := [[-x^2 + z\,y, [x + y + z - 1]]] \quad (9.8)$$

```
> rescad:=TTICADResCAD( PHI, [z,y,x], output=piecewise ):
  NumCellsInPiecewiseCAD(%);
```

```
Warning, The output from TTICADResCAD is only guaranteed truth
table invariant if no equational constraint is nullified by a
point in [y, x]
```

91 **(9.9)**

Note there is a cell [x=0, y=0, z free] over which f is zero but g changes sign!

We can avoid this error by using the full TTICAD algorithm, which splits the cell but is still much smaller than a sign-invariant CAD

```
> TTICAD( PHI, [z,y,x], output=piecewise): NumCellsInPiecewiseCAD
  (%);
  CADFull( [f,g], [z,y,x], output=piecewise, method=McCallum):
  NumCellsInPiecewiseCAD(%);
```

93

147 **(9.10)**

Example 3 - An equational constraint has main variable of a lower level than the other constraints

Consider

```
> f:=x: g:=x+y+1: PHI:=[ [f,[g]] ];
```

$$PHI := [[x, [y + x + 1]]] \quad \textbf{(9.11)}$$

```
> rescad:=TTICADResCAD( PHI, [y,x], output=piecewise ):
  NumCellsInPiecewiseCAD(%);
Warning, The output from TTICADResCAD is only guaranteed truth
table invariant if no equational constraint is nullified by a
point in [x]
Warning, An equational constraint will certainly be nullified
since x does not contain the main variable y.
Warning, there are no projection polynomials in [y, x]
```

3 **(9.12)**

Such cases are simple to recognize so an extra warning is provided

Note there is a cell [x=0, y free] over which f is zero but g changes sign. Once again, the TTICAD algorithm splits this

```
> TTICAD( [ [f,[g]] ], [y,x], output=piecewise):
  NumCellsInPiecewiseCAD(%);
  CADFull( [f,g], [y,x], output=piecewise, method=McCallum):
  NumCellsInPiecewiseCAD(%);
```

5

9 **(9.13)**

9. TTICAD Formulations and Heuristics

Consider

```
> f11:=y^2+x^2-1: f12:=x^3+y^3-1: g1:=y*x-1/4:
  f21:=(x-4)^2+(y-1)^2-1: f22:=(x-4)^3+(y-1)^3-1: g2:=(x-4)*(y-1)
  -1/4:
```

and the problem [f11=0 ∧ f12=0 ∧ g1<0] ∨ [f21=0 ∧ f22=0 ∧ g2<0].

We can solve this with a TTICAD

```
> PHI1:=[ [f11,[f12,g1]], [f21,[f22,g2]] ]:
  TTICAD(PHI1, [y,x] ): nops(%);
```

125 **(10.1)**

However there are different possible formulations for the TTICAD leading to CADs of different sizes, for example

```
> PHI2:=[ [f12,[f11,g1]], [f22,[f21,g2]] ]:
  TTICAD(PHI2, [y,x] ): nops(%);
```

85 **(10.2)**

We can obtain a list of different formulations using TTICADFormulations on a list of lists, each containing two lists - one of equational constraints and one of other constraints.

```
> Forms:=TTICADFormulations( [ [[f11,f12],[g1]], [[f21,f22],[g2]]
  ] ): nops(%);
```

16 **(10.3)**

There are 16 formulations because it there is not just the choice of equational constraints but also the choice of splitting QFFs. E.g.

```
> Forms[3];
```

$$\left[\left[y^2+x^2-1,\left[y\,x-\frac{1}{4}\right]\right],\left[y^3+x^3-1,[\,]\right],\left[y^2+x^2-2\,y-8\,x+16,\left[y^3+x^3-3\,y^2-12\,x^2+3\,y+48\,x-66,y\,x-x-4\,y+\frac{15}{4}\right]\right]\right] \quad \textbf{(10.4)}$$

Again, we can predict CAD complexity by running the measures on the projection factors

```
> TTICADProjFactors( PHI1, [y,x] ): sotd(%), ndrr(%,x);
  TTICADProjFactors( PHI2, [y,x] ): sotd(%), ndrr(%,x);
```

65, 14

101, 8 **(10.5)**

In this case the ndrr measure predicted correctly but the sotd measure did not. Emphasizing that these are just heuristics.

We can use the heuristic command to pick a formulation. Note that different heuristics may pick different formulations.

```
> TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x],
  heuristic=N): is(%=Forms[6]);
```

true **(10.6)**

```
> TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x],
  heuristic=S): is(%=Forms[1]);
```

true **(10.7)**

We use the default of N first then S to split. This also picks the sixth option, (which was PHI2 above).

```
> is(Forms[6]=expand(PHI2));
```

true **(10.8)**

We demonstrate the weighted heuristic. This considers the ranking by each heuristic, by default giving equal ratio to each. This can be varied with the Wratio argument which gives the ration ndrr:sotd (by default 1:1)

```
> TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x],
  heuristic=W, Wratio=[1,1]): is(%=Forms[2]);
```

$$true \tag{10.9}$$

The number of TTICAD formulations can be large so we offer two approaches to reducing the heuristic time.

```
> st:=time(): TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],
  [g2]]], [y,x]): et:=time()-st;
```

$$et := 0.303 \tag{10.10}$$

1: Ignore the possibility of splitting QFFs. Although this can help, it usually doesn't

```
> TTICADFormulations( [ [[f11,f12],[g1]], [[f21,f22],[g2]]],
  splitting=false ): nops(%);
```

$$4 \tag{10.11}$$

```
> st:=time(): TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],
  [g2]]], [y,x], splitting=false ): et:=time()-st;
```

$$et := 0.036 \tag{10.12}$$

2: Work on each QFF one by one - the so called modular approach. This will consider less possible formulations, and ignores the effect of the QFFs interacting.

```
> st:=time(): TTICADHeuristic( [ [[f11,f12],[g1]], [[f21,f22],
  [g2]]], [y,x], modular=true): et:=time()-st;
```

$$et := 0.036 \tag{10.13}$$

Note that we can also consider QFF formulations and heuristics individually ourselves

```
> TTICADQFFFormulations( [[f11,f12],[g1]] ): nops(%);
```

$$4 \tag{10.14}$$

```
> TTICADQFFHeuristic( [[f11,f12],[g1]], [y,x] );
```

$$\left[\left[y^3+x^3-1, \left[y^2+x^2-1, y\,x-\frac{1}{4}\right]\right]\right] \tag{10.15}$$

Note that we can give the input to TTICAD directly and it will use the default heuristic on each QFF to choose the formulation.

```
> TTICAD( [ [[f11,f12],[g1]], [[f21,f22],[g2]]], [y,x] ): nops(%)
  ;
```

$$85 \tag{10.16}$$

10. Projection polynomials vs lifting polynomials

When building a full sign-invariant CAD the set of projection polynomials is constructed during projection, and then used during lifting so that all the polynomials are sign-invariant on each cell. As noted above, this is not the case for ECCAD / TTICAD. Theorems on the reduced projection operators used for these algorithms allow us to (usually) ignore the non-equational constraints when lifting. Hence, while these polynomials are part of the projection set, they are not (usually) part of the lifting set.

The projection polynomials for these algorithms can be calculated independently using the commands **ECCADProjFactors** and **TTICADProjFactors.**
The lifting polynomials will vary from stack to stack. For all except the final lift the projection polynomials of that level will be used, as normal.
For the final lift it is usually only the equational constraints which are used. However, if an equational constraint is nullified on a zero dim cell, then the lifting set is extended for that stack only to include the corresponding non-equational constraints.
For information on the process change the TTICAD info level to 3..

```
> infolevel[TTICAD]:=3:
> f:=x*y: g:=x+y+1: PHI:=[ [f,[g]] ];
  TTICAD(PHI, [y,x]): nops(%);
```

$$PHI := [[y\,x, [y + x + 1]]]$$

```
TTI_TTIGenerateStack: the equational constraint factor y*x  is
nullified on the cell  [[4] [regular_chain [[0 0]]]]
TTI_TTIGenerateStack: the cell is zero-dimensional so we can
continue by expanding the lifting set on this cell.
```

17 **(11.1)**

The same process is used for ECCAD:

```
> ECCAD([f,[g]], [y,x] ): nops(%);
TTI_TTIGenerateStack: the equational constraint factor y*x  is
nullified on the cell  [[4] [regular_chain [[0 0]]]]
TTI_TTIGenerateStack: the cell is zero-dimensional so we can
continue by expanding the lifting set on this cell.
```

17 **(11.2)**

```
> infolevel[TTICAD]:=1:
```

Modifying the lifting set in this way for ECCAD follows from the original theorems but does not appear to have been implemented before. For example, in QEPCAD the full projection set is used for lifting.
Consider the following example:

```
> f1:=x^2+y^2-1: g1:=x*y-1/4:
  f2:=(x-4)^2+(y-1)^2-1: g2:=(x-4)*(y-1)-1/4:
> ECCAD( [f1*f2, [f1,f2,g1,g2]], [y,x] ): nops(%);
```

145 **(11.3)**

Using QEPCAD with f1*f2 declared an EC produces 249 cells.
The reason is that g1 and g2 are included in the lifting set for QEPCAD, but never here.
We can simulate the QEPCAD approach with the optional command LiftAll

```
> ECCAD( [f1*f2, [f1,f2,g1,g2]], [y,x], LiftAll=true ): nops(%);
```

249 **(11.4)**

This can also be used with TTICADs.

```
> PHI:=[ [f1,[g1]], [f2,[g2]] ]:
  TTICAD( PHI, [y,x] ): nops(%);
  TTICAD( PHI, [y,x], LiftAll=true ): nops(%);
```

105

185 **(11.5)**

Note that there is no theoretical reason to use LiftAll. It is only included here to make comparison with QEPCAD easier.

11. Choosing a Variable ordering

The variable ordering can make a difference to size and computation time of a CAD.

```
> f:=(x-1)*(y^2+1)-1:
  CADFull( {f}, [y,x], method=Collins): nops(%);
  CADFull( {f}, [x,y], method=Collins): nops(%);
```

11

3 **(12.1)**

The VariableOrderings command returns a list of the possible variable orderings for a list of variables

```
> VariableOrderings( [y,x] );
```

$$[[y, x], [x, y]] \quad \textbf{(12.2)}$$

For many applications there are restrictions on variable orderings. For example, in quantifier elimination all quantified variables must be projected first, and two adjacent quantified variables can only be switched in the orderings if they have the same quantifier. The VariableOrderings command can consider such situations when given a list of lists of variables (representing variable blocks within which different orderings are permitted.

```
> VariableOrderings( [x,y,z,w] ): nops(%);
  VariableOrderings( [[x,y], [z,w]] ); nops(%);
```

24

$$[[x, y, z, w], [y, x, z, w], [x, y, w, z], [y, x, w, z]]$$

4 **(12.3)**

```
> VariableOrderings( [w,x,y,z,a,b] ): nops(%);
  VariableOrderings( [[w],[x,y,z], [a,b]] ); nops(%);
```

720

$$[[w, x, y, z, a, b], [w, x, z, y, a, b], [w, y, x, z, a, b], [w, y, z, x, a, b], [w, z, x, y, a, b], [w, z, y, x, a, b], [w, x, y, z, b, a], [w, x, z, y, b, a], [w, y, x, z, b, a], [w, y, z, x, b, a], [w, z, x, y, b, a], [w, z, y, x, b, a]]$$

12 **(12.4)**

We can use the ndrr and sotd measures applied to projection polynomials to pick the best variable ordering for a problem.
This is encoded in VariableOrderingHeuristic. The first entry should be a list of variables (or list of lists as above). The second should be the input to the CAD algorithm. The output is then the suggested ordering.

```
> VariableOrderingHeuristic( [y,x], {f} );
```

$$[x, y] \quad \textbf{(12.5)}$$

The algorithm assumes a sign-invariant CAD (i.e. using CADFull) but this can be specified, (along with the method if using CADFull).

```
> VariableOrderingHeuristic( [y,x], {f}, algorithm=CADFull,
  method=McCallum );
```

$$[x, y] \quad \textbf{(12.6)}$$

Note the usual heuristic options

```
> VariableOrderingHeuristic( [y,x], {f}, algorithm=CADFull,
  heuristic=S );
```

```
Warning, There are 2 formulations which heuristic S cannot
differentiate.  Only the first lexicographically has been
returned.  To display them all run with option SeeAll=true or
```

to supress this message run with SeeAll=false.

$$[x, y] \tag{12.7}$$

```
> VariableOrderingHeuristic( [y,x], {f}, algorithm=CADFull,
  heuristic=N );
```

$$[x, y] \tag{12.8}$$

We can use this command to choose the best variable ordering for ECCAD and TTICAD as well. In general she second argument should be input suitable for the respective algorithm.

```
> f:=x^2+y^2-1: g:=x*(y-1/2):
  ECCAD( [f, [g]], [y,x]): nops(%);
  ECCAD( [f, [g]], [x,y]): nops(%);
```

$$43$$
$$23 \tag{12.9}$$

```
> VariableOrderingHeuristic( [y,x], [f, [g]], algorithm=ECCAD );
```

$$[x, y] \tag{12.10}$$

```
> f1:=x^2+y^2-1: g1:=x*y-1/4:
  f2:=x^2-y^2-1: g2:=(x-4)*(y-1)-1/4:
  PHI:=[ [f1,[g1]], [f2,[g2]] ]:
  TTICAD( PHI, [y,x] ): nops(%);
  TTICAD( PHI, [x,y] ): nops(%);
```

$$101$$
$$175 \tag{12.11}$$

12. Greedy algorithms for variable ordering choices

We have implemented greedy algorithms for choosing variable orderings. This is where each variable is chosen in turn based on the projection polynomials calculated thus far.
For example, the quartic problem: x is quantified so comes first regardless. Then there are 6 variable orderings

```
> ord:=[p,q,r]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[p,r,q]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[q,p,r]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[q,r,p]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[r,p,q]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
  ord:=[r,q,p]: CADFull( [x^4+p*x^2+q*x+r], [x,op(ord)], method=
  McCallum ): ord,nops(%);
```

$$[p, q, r], 333$$
$$[p, r, q], 333$$
$$[q, p, r], 277$$
$$[q, r, p], 321$$
$$[r, p, q], 177$$
$$[r, q, p], 213 \qquad \textbf{(13.1)}$$

So the best ordering in terms of cell count is [x,r,p,q].
The standard heuristic find this:

```
> st:=time(): VariableOrderingHeuristic( [[x],[p,q,r]], [x^4+p*
  x^2+q*x+r], heuristic=S, algorithm=CADFull); et:=time()-st;
```

$$[x, r, p, q]$$
$$et := 0.029 \qquad \textbf{(13.2)}$$

If we use the greedy algorithm we also find this, but quicker.

```
> st:=time(): VariableOrderingHeuristic( [[x],[p,q,r]], [x^4+p*
  x^2+q*x+r], heuristic=S, algorithm=CADFull, greedy=true); et:=
  time()-st;
```

$$[x, r, q, p]$$
$$et := 0.009 \qquad \textbf{(13.3)}$$

But note that it was a lot quicker. For larger problems the speed up is more important.

Greedy algorithms are also present for picking the variable orderings for ECCAD:

```
> f:=x^2+y^2+z^2-1: g1:=x*(y-1/2): g2:=z*(y+1/2):
> ord:=[x,y,z]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[x,z,y]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[y,x,z]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[y,z,x]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[z,x,y]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
  ord:=[z,y,x]: ECCAD( [f, [g1,g2]], ord): ord,nops(%);
```

$$[x, y, z], 187$$
$$[x, z, y], 131$$

$$[y, x, z], 237$$
$$[y, z, x], 237$$
$$[z, x, y], 131$$
$$[z, y, x], 187 \quad \textbf{(13.4)}$$

The normal heuristic picks the best two

```
> st:=time(): VariableOrderingHeuristic( [x,y,z], [f,[g1,g2]],
  heuristic=NS, algorithm=ECCAD, SeeAll=true); et:=time()-st;
```

$$[[x, z, y], [z, x, y]]$$
$$et := 0.028 \quad \textbf{(13.5)}$$

The greedy method picks the joint best, and quicker.

```
> st:=time(): VariableOrderingHeuristic( [x,y,z], [f,[g1,g2]],
  heuristic=S, algorithm=ECCAD, greedy=true); et:=time()-st;
```

$$[z, x, y]$$
$$et := 0.008 \quad \textbf{(13.6)}$$

We can also use greedy algorithms for variable blocks:

```
> st:=time(): VariableOrderingHeuristic( [[x],[y,z]], [f,[g1,g2]
  ], heuristic=S, algorithm=ECCAD, SeeAll=true); et:=time()-st;
  st:=time(): VariableOrderingHeuristic( [[x],[y,z]], [f,[g1,g2]
  ], heuristic=S, algorithm=ECCAD, greedy=true); et:=time()-st;
```

$$[x, z, y]$$
$$et := 0.009$$
$$[x, z, y]$$
$$et := 0.005 \quad \textbf{(13.7)}$$

```
> st:=time(): VariableOrderingHeuristic( [[x,y],[z]], [f,[g1,g2]
  ], heuristic=S, algorithm=ECCAD, SeeAll=true); et:=time()-st;
  st:=time(): VariableOrderingHeuristic( [[x,y],[z]], [f,[g1,g2]
  ], heuristic=S, algorithm=ECCAD, greedy=true); et:=time()-st;
```

$$[x, y, z]$$
$$et := 0.009$$
$$[x, y, z]$$
$$et := 0.004 \quad \textbf{(13.8)}$$

Finally we consider greedy algorithms for picking the variable ordering for TTICAD:

```
> f1:=x^2+y^2+z^2-1: g1:=x*y*z-1/4: f2:=x^2-y^2-z^2-1: g2:=(x-4)*
  (y-1)-1/4+z:
  PHI:=[ [f1,[g1]], [f2,[g2]] ]:
> Digits:=20:
  ord:=[x,y,z]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[x,z,y]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[y,x,z]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[y,z,x]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[z,x,y]: TTICAD( PHI, ord): ord,nops(%);
  ord:=[z,y,x]: TTICAD( PHI, ord): ord,nops(%);
  Digits:=10:
```

$$[x, y, z], 1497$$
$$[x, z, y], 889$$
$$[y, x, z], 1135$$
$$[y, z, x], 351$$
$$[z, x, y], 985$$

$$[z, y, x], 463 \tag{13.9}$$

```
> st:=time(): VariableOrderingHeuristic( [x,y,z], PHI, heuristic=
  S, algorithm="TTICAD", SeeAll=true); et:=time()-st;
```

$$[x, z, y]$$

$$et := 0.067 \tag{13.10}$$

```
> st:=time(): VariableOrderingHeuristic( [x,y,z], PHI, heuristic=
  S, algorithm="TTICAD", greedy=true); et:=time()-st;
```

$$[z, y, x]$$

$$et := 0.016 \tag{13.11}$$

Greedy method actually picks better in this example (for sotd). But with the non-greedy approach we can make use of ndrr as well and get the very best (albeit at greater running cost).

```
> st:=time(): VariableOrderingHeuristic( [x,y,z], PHI, heuristic=
  NS, algorithm="TTICAD", SeeAll=true); et:=time()-st;
```

$$[y, z, x]$$

$$et := 0.117 \tag{13.12}$$

13. Layered sub-CADs

A *layer* of a CAD consists of cells of a given dimension. An *l-Layered sub-CAD (l-LCAD)* consists of the top *l* layers of a given CAD of $\mathbb{R}^n$ which is those cells with dimension *(n-l+1)* to *n*. The LCAD commands offer various ways to compute such sub-CADs.

Direct Layered sub-CADs

If you know how many layers of cells are required simple use the **LCAD** command with input: a list of polynomials, a positive integer specifying the number of layers required, and an ordered list of variables. For example:

```
> LCAD([x^2+y^2+z^2-1], 2, [z,y,x]): nops(%);
```

Warning, no method was specified, McCallum's algorithm will be used

17 **(14.1.1)**

As can be seen above, if no method is specified a warning message appears and McCallum's algorithm is used as default. We can specify the projection operator between `McCallum` and `Collins`, which can make a difference in cell counts, as shown below.

```
> F:=[x^2*y^2-z^3+1,x+y+z-1]: vars:=[z,y,x]:
  LCAD(F, 1, vars, method=McCallum): nops(%);
  LCAD(F, 1, vars, method=Collins): nops(%);
```

24

48 **(14.1.2)**

The output format can also be specified as either *list*, `listwithrep`, or *piecewise*.

```
> LClist   := LCAD(F, 1, vars, method=McCallum, output=list):
  LClist[1];
  LClistwr := LCAD(F, 1, vars, method=McCallum, output=
  listwithrep): LClistwr[1];
```

$$[[1, 1, 1], [regular\_chain, [[-9, -9], [-45, -45], [53, 53]]]]$$

$$\left[[1, 1, 1], \left[x < RootOf\left(4\_Z^9 + 15\_Z^8 - 96\_Z^7 + 162\_Z^6 - 72\_Z^5 + 45\_Z^4 - 108\_Z^3 + 54\_Z^2 + 27, index = real_1\right), y < RootOf\left(\_Z^3 + \left(x^2 + 3x - 3\right)\_Z^2 + \left(3x^2 - 6x + 3\right)\_Z + x^3 - 3x^2 + 3x, index = real_1\right), z < RootOf\left(-y^2x^2 + \_Z^3 - 1, index = real_1\right)\right], [regular\_chain, [[-9, -9], [-45, -45], [53, 53]]]\right] \quad \textbf{(14.1.3)}$$

Displaying Layered sub-CADs

It can often be difficult to visualise a CAD. The piecewise construct in Maple allows for a more intuitive display. We emulate the piecewise output within layered sub-CADs by using the LCADDisplay procedure. This requires a sub-CAD given in the 'listwithrep' format. **`LCADDisplay`** then produces a tree-like structure containing all cells in a layered sub-CAD. It also indicates the *terminating sections* of a sub-CAD by displaying where branches have been terminated.

Example: A 1-layered sub-CAD of the circle is produced, identifying five two-dimensional cells: the region $x<-1$, the regions between $x=-1$ and $x=1$ above and below the circle, the interior of the circle, and the region $x>1$.

```
> LCAD([x^2+y^2-1],1,[y,x], method=McCallum, output=
  listwithrep): nops(%);
  LCADDisplay(%%);
```

5

$$\left\{\begin{array}{ll} [regular\_chain, [[-2,-2],[0,0]]] & x<-1 \\ ["**********"] & branch = truncated \\ \left\{\begin{array}{ll} [regular\_chain, [[0,0],[-2,-2]]] & y<-\sqrt{-x^2+1} \\ ["**********"] & branch = truncated \\ [regular\_chain, [[0,0],[0,0]]] & -\sqrt{-x^2+1}<y<\sqrt{-x^2+1} \\ ["**********"] & branch = truncated \\ [regular\_chain, [[0,0],[2,2]]] & \sqrt{-x^2+1}<y \\ ["**********"] & branch = truncated \end{array}\right. & -1<x<1 \\ [regular\_chain, [[2,2],[0,0]]] & 1<x \end{array}\right. \quad \textbf{(14.2}$$

If we consider another layer we see that some of the missing parts have been filled in. Note also that we can also just pass piecewise as an output option.

```
> LCAD([x^2+y^2-1],2,[y,x], method=McCallum, output=
  listwithrep): nops(%);
```

11 **(14.2.2)**

```
> #LCAD([x^2+y^2-1],2,[y,x], method=McCallum, output=piecewise)
  ;
```

There are still cells missing (namely the points *(-1,0)* and *(1,0)*).

```
> LCAD([x^2+y^2-1],3,[y,x], method=McCallum, output=
  listwithrep): nops(%);
```

13 **(14.2.3)**

```
> #LCAD([x^2+y^2-1],3,[y,x], method=McCallum, output=piecewise)
  ;
```

Of course, this is now the entire CAD, and the same output as:

```
> CADFull( [x^2+y^2-1],[y,x], method=McCallum, output=
  piecewise);
```

$$\left[\mathit{regular\_chain}, [[-2,-2],[0,0]]\right] \quad x<-1$$

$$\begin{cases} [\mathit{regular\_chain}, [[-1,-1],[-1,-1]]] & y<0 \\ [\mathit{regular\_chain}, [[-1,-1],[0,0]]] & y=0 \\ [\mathit{regular\_chain}, [[-1,-1],[1,1]]] & 0<y \end{cases} \quad x=-1$$

$$\begin{cases} [\mathit{regular\_chain}, [[0,0],[-2,-2]]] & y<-\sqrt{-x^2+1} \\ [\mathit{regular\_chain}, [[0,0],[-1,-1]]] & y=-\sqrt{-x^2+1} \\ [\mathit{regular\_chain}, [[0,0],[0,0]]] & -\sqrt{-x^2+1}<y<\sqrt{-x^2+1} \\ [\mathit{regular\_chain}, [[0,0],[1,1]]] & y=\sqrt{-x^2+1} \\ [\mathit{regular\_chain}, [[0,0],[2,2]]] & \sqrt{-x^2+1}<y \end{cases} \quad -1<x<1 \quad \textbf{(14.2.4)}$$

$$\begin{cases} [\mathit{regular\_chain}, [[1,1],[-1,-1]]] & y<0 \\ [\mathit{regular\_chain}, [[1,1],[0,0]]] & y=0 \\ [\mathit{regular\_chain}, [[1,1],[1,1]]] & 0<y \end{cases} \quad x=1$$

$$\left[\mathit{regular\_chain}, [[2,2],[0,0]]\right] \quad 1<x$$

Recursive Layered sub-CADs

The **LCADRecursive** command returns a layered sub-CAD and an inert command that can be evaluated to return the next layer of the sub-CAD. This command is useful if you do not know in advance how many layers are required and which to compute new layers incrementally without repeating computations.

The command requires four inputs: a list of polynomials, a list of variables and two lists of cells. The first of these is a list of *terminating sections* (cells where a previous call to `LCADRecursive` halted) and the second is the previously computed layered sub-CAD. If the user inputs empty lists for these then the command will produce the initial 1-layered sub-CAD.

The output of the function is two-fold: a layered sub-CAD containing one more layer than the one in the fourth input, and an inert **LCADRecursive** call which can be used to build the next layer.

For example:

```
> LcadA, RecCallA := LCADRecursive([x^2+y^2+z^2-1], [x,y,z],
  [], [], method=McCallum, output=listwithrep):
  nops(LcadA); #LCADDisplay(LcadA);
```

7 **(14.3.1)**

If we examine the inert call we see it is a complete **LCADRecursive** call, preceded by the `%` symbol (a Maple standard to render a function call inert). Stored within the call are the original input polynomials and variables, the terminating sections and layered sub-CAD of the previous call, and any options specified.

```
> op(0,RecCallA);
```

```
#op(RecCallA);
```

$$\%LCADRecursive \qquad (14.3.2)$$

To evaluate the inert call, you use Maple's **value** command.

```
> LcadB, RecCallB := value(RecCallA):
  nops(LcadB);
```

$$17 \qquad (14.3.3)$$

We can repeat this until the entire CAD is calculated

```
> LcadC, RecCallC := value(RecCallB):
  nops(LcadC);
```

$$23 \qquad (14.3.4)$$

```
> LcadD, RecCallD := value(RecCallC):
  nops(LcadD);
```

$$25 \qquad (14.3.5)$$

Of course, this is the same output as:

```
> CADFull([x^2+y^2+z^2-1], [x,y,z], method=McCallum, output=
  listwithrep): nops(%);
```

$$25 \qquad (14.3.6)$$

Computing the sub-cad recursively is comparable in terms of efficiency to computing it directly

```
> F:=[z^2+w^2-2,z^3-w+2]: vars:=[z,w]:
  s0:=time(): cad1 := CADFull(F,vars,method=McCallum): s1:=time
  ()-s0; nops(cad1);
```

$$s1 := 0.244$$

$$53 \qquad (14.3.7)$$

```
> F:=[u^2+v^2-2,u^3-v+2]: vars:=[u,v]:
  t0:=time():
  LC1,RC1:=LCADRecursive(F,vars,[],[],method=McCallum,
  resetproj=true):
  LC2,RC2:=value(RC1):
  LC3,RC3:=value(RC2):
  t1:=time()-t0; nops(LC3);
```

$$t1 := 0.276$$

$$53 \qquad (14.3.8)$$

A note on timings in Maple

It can be difficult to compare routines in Maple as Maple will often reuse previous results to speed up function calls. Hence a second call computing the same CAD or sub-CAD can seem quicker than it should, as below:

```
> F:=[x^2+y^2-2,x^3-y+2]: vars:=[x,y]:

  t0:=time():
  LC1,RC1:=LCADRecursive(F,vars,[],[],method=McCallum,
  resetproj=true):
  LC2,RC2:=value(RC1):
  LC3,RC3:=value(RC2):
  t1:=time()-t0; nops(LC3);

  s0:=time(): cad1 := CADFull(F,vars,method=McCallum): s1:=
  time()-s0; nops(cad1);
```

$$t1 := 0.271$$

$$53$$

$$s1 := 0.167$$

$$53 \qquad \textbf{(14.3.1.1)}$$

Hence in the demonstration above we used different variables to avoid this. The optimal approach would actually be to start separate Maple sessions for each computation!

Optionally, the **LCADRecursive** procedure can store the projection polynomials in a global variable so that they do not need to be recomputed for each call. The only correctness check performed is that the input polynomials are a subset of these, which can be misled. Hence the default behavior is to recompute each time. This can be overridden using ***resetproj=false***.

```
> Lcad,RecCall:=LCADRecursive([x*y*z-1/4],[x,y,z],[],[],
  resetproj=true):
  nops(Lcad);
```

$$8 \qquad \textbf{(14.3.9)}$$

```
> infolevel[ProjectionCAD]:=3:
  F:=[u^2+v^2-2,u^3-v+2]: vars:=[u,v]:
  LC1,RC1:=LCADRecursive(F,vars,[],[],method=McCallum,
  resetproj=false): nops(LC1);
  LC2,RC2:=value(RC1): nops(LC2);
  LC3,RC3:=value(RC2): nops(LC3);
  infolevel[ProjectionCAD]:=1:
LCAD_Recursive: Projection polynomials to be reset as no
existing cells passed in.
CADFull: produced set of 4 projection factors using the
McCallum algorithm.
PCAD_ProjCADLift: produced CAD of [v] -space with 11 cells
                                18
LCAD_Recursive: Projection polynomials left as defined.
                                44
LCAD_Recursive: Projection polynomials left as defined.
                                53
```

(14.3.10)

Layered sub-TTICADs

It is possible to create layered TTICADs, by using the TTICAD projection operator, creating the appropriate layered *(n-1)*-dimensional sub-CAD, and lifting over using TTICAD lifting whilst respecting cell dimensions. This is implemented in the **LTTICAD** command, which takes a list of quantifier free formulae, with designated equational constraints, in the same format as the **TTICAD** command, along with the number of required layers and ordered variables.

```
> PHI := [ [x^2+y^2+z^2-1,[x*y*z-1/4]], [(x-1)^2+(y-1)^2+(z-1)
  ^2-1,[(x-1)*(y-1)*(z-1)-1/4]] ]; vars:=[x,y,z]:
```

$$PHI := \left[\left[x^2+y^2+z^2-1, \left[x\,y\,z-\frac{1}{4}\right]\right], \left[(x-1)^2+(y-1)^2+(z-1)^2-1, \left[(x-1)\,(y-1)\,(z-1)-\frac{1}{4}\right]\right]\right] \qquad \textbf{(14.4.1)}$$

```
> Digits:=20:
  st:=time():
  LTTICAD(PHI,1,vars): N:=nops(%);
  et:=time()-st: print("1 Layer: ", et, N);
```

```
st:=time():
LTTICAD(PHI,2,vars): N:=nops(%);
et:=time()-st: print("2 Layer: ", et, N);
st:=time():
LTTICAD(PHI,3,vars): N:=nops(%);
et:=time()-st: print("3 Layer: ", et, N);
st:=time():
LTTICAD(PHI,4,vars): N:=nops(%);
et:=time()-st: print("4 Layer: ", et, N);
Digits:=10:
```

$$N := 93$$

"1 Layer: ", 3.336, 93

$$N := 299$$

"2 Layer: ", 7.210, 299

$$N := 455$$

"3 Layer: ", 7.971, 455

$$N := 497$$

"4 Layer: ", 7.985, 497 **(14.4.2)**

```
> Digits:=20: st:=time(): TTICAD(PHI,vars): N:=nops(%): et:=
  time()-st: print("Full TTTICAD: ", et, N); Digits:=10:
```

"Full TTTICAD: ", 7.764, 497 **(14.4.3)**

Cell Distributions in CADs/TTICADs

It can be useful to look at the distribution of cell dimensions within CADs. Our experiments suggest CADs often have cell dimensions roughly in a binomial distribution with high p value. This can be used to predict the size of final CADs by considering the 1-layered sub-CAD (which is considerably easier to compute). To see the total number of cells of each dimension (from 0 to n) use the **CADDist** command.

```
> CADDist([x^2+y^2-1],[x,y]);
```

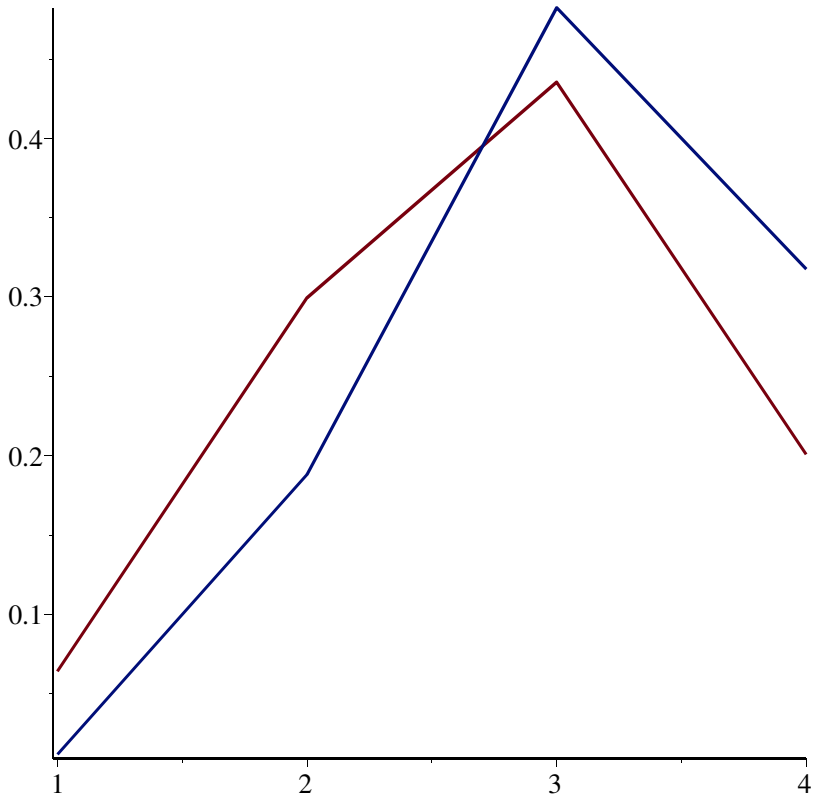
$$[2, 6, 5]$$ **(14.5.1)**

To get an easier feel for comparing how the cells are distributed between different CADs, the normed distribution can be computed with the **LCADNormDist** command, which can then be plotted on the same scale.

```
> NormDist1 := CADNormDist([x^2+y+z-1,x*z-1,x*y-1],[x,y,z]);
  NormDist2 := CADNormDist([x^2+y+z-1,x*z-1,x*y-1],[z,y,x]);
  plot([[seq([i,NormDist1[i]],i=1..nops(NormDist1))],[seq([i,
  NormDist2[i]],i=1..nops(NormDist2))]]);
```

$$NormDist1 := \left[\frac{76}{1185}, \frac{71}{237}, \frac{172}{395}, \frac{238}{1185}\right]$$

$$NormDist2 := \left[\frac{1}{85}, \frac{16}{85}, \frac{41}{85}, \frac{27}{85}\right]$$



There are analogous commands for TTICADs: `TTIDist` and **`TTINormDist`.**

```
> TTICADDist([[x^2+y-1,[x+y-1]]],[x,y]);
  TTICADNormDist([[x^3+y^2-1,[x*y+y^2-1,x^3-1]],[y^3-x,[x^2-y]]
  ],[x,y]);
```

$$[3, 9, 7]$$

$$\left[\frac{11}{63}, \frac{31}{63}, \frac{1}{3}\right] \qquad \textbf{(14.5.2)}$$

14. Variety sub-CADs

When a problem contains an equational constraint (EC), an equation that must be satisfied for a solution, we can limit our output to the variety it defines. We call this a *Variety sub-CAD (VCAD)* Suppose the EC is f=0. Our implementations is restricted to the case where all factors of *f* contain the main variable.

VCADs

We can produce a variety sub-CAD using the **VCAD** command. This uses the **ECCAD** projection operator and lifting procedure. However, it will only return the sections where *f=0*. The input is the same as **ECCAD** which takes a list consisting of the equational constraint and a list of non-equational constraint polynomials.

```
> VCAD( [x^2+y^2-1,[x*y-1/4,x^3-y^2]],[x,y]): nops(%);
  ECCAD( [x^2+y^2-1,[x*y-1/4,x^3-y^2]],[x,y]): nops(%);
  CADFull([x^2+y^2-1, x*y-1/4,x^3-y^2], [x,y],method=McCallum):
  nops(%);
```

28
73
161 **(15.1.1)**

Variety sub-TTICADs

We can also create a Variety sub-TTICAD. If each quantifier free formula has equational constraint f_i=0, then **VTTICAD** returns the cells satisfying $\prod_{i=1}^{k} f_i = 0$.

```
> Digits:=20:
  PHI := [[x^2+y^2+z^2-1,[x*y*z-1/4]],[(x-1)^2+(y-1)^2+(z-1)^2
  -1,[(x-1)*(y-1)*(z-1)-1/4]]]; vars:=[x,y,z]:
  VTTICAD(PHI,vars): nops(%);
  TTICAD(PHI,vars): nops(%);
  Digits:=10:
```

$$PHI := \left[\left[x^2+y^2+z^2-1, \left[x\,y\,z-\frac{1}{4}\right]\right], \left[(x-1)^2+(y-1)^2+(z-1)^2-1, \left[(x-1)\,(y-1)\,(z-1)-\frac{1}{4}\right]\right]\right]$$

176
497 **(15.2.1)**

Layered variety sub-CAD

We can combine layered and variety sub-CADs. For an input with equational constraint $f_i = 0$, an *l-layered variety sub-CAD* consists of the top l layers of cells on the variety defined by $f_i = 0$.

To compute a layered variety sub-CAD the **LVCAD** command can be used. This takes a list containing an equational constraint and a list of non-equational constraints, along with the number of layers required and a list of variables.

```
> LVCAD([x^2+y^2-1,[x*y-1/4,x^3-y^2]],1,[x,y]): nops(%);
```

$$14 \qquad \textbf{(15.3.1)}$$

Finally, we can combine all available technology to produce a *layered variety sub-TTICAD (LVTTICAD)*. The **LVTTICAD** command takes the same input as a TTICAD along with the number of layers required.

```
> PHI := [[x^2+y^2+z^2-1,[x*y*z-1/4]],[(x-1)^2+(y-1)^2+(z-1)^2
  -1,[(x-1)*(y-1)*(z-1)-1/4]]]; vars:=[x,y,z]:
```

$$PHI := \left[\left[x^2+y^2+z^2-1, \left[x\,y\,z-\frac{1}{4}\right]\right], \left[(x-1)^2+(y-1)^2+(z-1)^2-1, \left[(x-1)\,(y-1)\,(z-1)-\frac{1}{4}\right]\right]\right] \qquad \textbf{(15.3.2)}$$

```
> Digits:=20:
  LVTTICAD(PHI,1,vars): nops(%);
  LTTICAD(PHI,1,vars): nops(%);
  TTICAD(PHI,vars): nops(%);
  Digits:=10:
```

$$48$$
$$93$$
$$497 \qquad \textbf{(15.3.3)}$$

15. Avoiding theoretical failure with Sub-CAD

Recall the example from Section 3

```
> f:=a*e+b*d+c*e+d+e; vars:=[a,b,c,d,e];
```

$$f := a\,e + b\,d + c\,e + d + e$$

$$vars := [a, b, c, d, e] \quad \textbf{(16.1)}$$

```
> CADFull( [f], vars, method=McCallum  ): nops(%);
Warning, The input is not well-oriented (there is
nullification on cell [1, 2, 2]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [2, 2, 1]).  The output cannot be
guaranteed correct.
Warning, The input is not well-oriented (there is
nullification on cell [3, 4, 2]).  The output cannot be
guaranteed correct.
```

241 **(16.2)**

The output could not be guarenteed correct (i.e. sign-invariant) as the input failed a technical condition known as well-orientedness.
For the condition to occur at all is rare, and when it does, it likely only affects a small part of the CAD.
Hence in cases where a CAD cannot be computed due to this theoretical failure, a sub-CAD is often still possible.

```
> LCAD([f],1,[a,b,c,d,e], method=McCallum,failure=err): nops(%);
```

48 **(16.3)**

```
> LCAD([f],2,[a,b,c,d,e], method=McCallum,failure=err): nops(%);
```

148 **(16.4)**

```
> LCAD([f],3,[a,b,c,d,e], method=McCallum, failure=err):
Error, (in LCAD ProjCADLiftNLayered) there is nullification on
cell [1, 2, 2] which has dim>0. The outputted CAD cannot be
guaranteed sign-invariant.
```

We see with this example that a 1 or 2 layered sub-CAD does not trigger the message. Hence we can get descriptions of the cells of full dimension, and the cells of one dimension lower without any problem.

In theory we could also avoid such failure with variety sub-CADs. However, our present implementation only works with varieties with all components of full dimension: lifting to these does not carry a risk of failure (as it is the final lift so only sign-nvariance is required).